Away From `co_yield` For `std::execution::task`

Document Number: P4282R0
Date: 2026-06-27
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper updates the design and wording of `std::execution::task` given the adoption of P3950 [1].

Background

The coroutine promise protocol shipped in C++20 not only banned `return_void` and `return_value` on the same promise type, but did so in an unusually restrictive manner [1]. Notably the aforementioned manner barred implementing the aforementioned mutual exclusion via mutually exclusive constraints.

P3950 [1] proposed changing the above. Not only allowing for `return_void` and `return_value` to be constrained such that they are mutually exclusive, but also allowing them to both be simultaneously valid.

`std::execution::task` [2] allows a coroutine to end in error without throwing an exception by yielding an instance of an instantiation of `std::execution::with_error`.

While `std::execution::task` supports stopped completion signals the promise type has no bona fide manner in which the coroutine body can emit them. Instead authors of coroutine bodies must use `co_await std::execution::just_stopped()`.

Discussion

`co_return std::execution::with_error(...)`

`co_yield` is a mechanism for coroutines to produce an intermediate value for consumption by its caller. While it is not guaranteed that a coroutine resume after `co_yield` (since `std::coroutine_handle::destroy` exists) it's arguable that the general, idiomatic expectation is that it does resume thereafter (consider the design of `std::generator` [3]).

`co_return` is a mechanism for ending a coroutine. A coroutine does not resume after `co_return` (this is implicit in the formulation of the "replacement body" (§9.6.4 [dcl.fct.def.coroutine]) in terms of which coroutines are specified).

For support of the above, consider the simplicity of the proposed wording (later in this paper) for the overload of `std::execution::task::promise_type::return_value` which accepts an instance of an instantiation of `std::execution::with_error`. Contrast this simplicity with the complexity of the wording for the value returned by `std::execution::task::promise_type::yield_value` (§33.13.6.5 [task.promise]):

*"An awaitable object of unspecified type whose member functions arrange for the calling coroutine to be suspended and then completes the asynchronous operation associated with STATE(*this). Let st be a reference to `STATE(*this)`. Then the asynchronous operation completes by first destroying the coroutine frame using `st.handle.destroy()` and then invoking `set_error(std::move(st.rcvr), Cerr(std::move(err.error)))`."*

The reason for this stark contrast is simple: `co_return` is designed to end coroutines, `co_yield` is not.

From within a coroutine body one can interact the promise via `co_return` (or flowing off the end of the coroutine body which is specified as being equivalent to `co_return;` in the instances in which it does not constitute undefined behavior (§8.8.5 [stmt.return.coroutine])), `co_yield`, and `co_await`, therefore `std::execution::task` must use one of them to communicate non-exception error types to the promise.

Given the reasoning above `co_return` seems like a better fit than `co_yield` because sending an error completion signal is terminal (i.e. it ought to cause the coroutine to end). However this would require the promise type for `void` tasks to have a `return_value` member function simultaneously with a `return_void` member function. This was banned before P3950 and therefore `co_yield` was the only viable, general option.

P3950 has now been accepted into the C++29 working draft and therefore the above can be revisited. This paper therefore proposes enriching `std::execution::task::promise_type` with a `return_value` overload which accepts `std::execution::with_error` and which functions equivalently to `std::execution::task::promise_type::yield_value`.

`co_return std::execution::with_stopped()`

Previously if the author of a coroutine body which uses `std::execution::task` wished to end the asynchronous operation with stopped they had to write `co_await std::execution::just_stopped()`. This is misleading for similar reasons as `co_yield` std::execution::with_error(...) (see above): Both co_await and co_yield ordinarily indicate that execution of the coroutine body will eventually resume, whereas co_await std::execution::just_stopped() will never resume (note that this applies to any asynchronous operation co_awaited from within a coroutine, not just std::execution::set_stopped, i.e. if that operation completes with std::execution::set_stopped the coroutine awaiting thereupon will never resume).

Just as for co_yield std::execution::with_error(...) there is a solution to the above in a post-P3950 world: co_return an instance of a tag type which indicates that the asynchronous operation should end with stopped. Therefore this paper proposes the addition of std::execution::with_stopped which may be used to indicate that a coroutine ought to end with std::execution::set_stopped via co_return std::execution::with_stopped().

co_return std::execution::with_value(...)

The trichotomous nature of std::execution completion dispositions (i.e. set_value, set_stopped, and set_error) might lead one to believe that if std::execution::with_error and ::with_stopped exist, then so too should std::execution::with_value. While this feels intuitive we should ask what purpose a hypothetical std::execution::with_value would serve. Two possibilities spring to mind:

- Disambiguating to allow co_return std::execution::with_value(std::execution::with_error(...)) and/or co_return std::execution::with_value(std::execution::with_stopped{})
- Allowing std::execution::task to generate value completions with arities higher than one

The second bullet can be immediately dismissed: No instantiation of std::execution::task leads to the advertisement of any value completions with an arity higher than one. Bearing in mind that T is the first template argument to std::execution::task consider how the value completion thereof is computed (§33.13.6.2 [task.class]):

"set_value_t() if T is void, and set_value_t(T) otherwise[.]"

Therefore co_return std::execution::with_value(v_0, ..., v_n) would serve no purpose since it would be unable to correspond to a value completion.

All that remains then is to consider the first bullet from above. Such disambiguation only has utility if we expect std::execution::with_error and std::execution::with_stopped to be used in contexts other than directly as an argument to co_return. We have no such expectation, and that is not part of the design intention of these types. Therefore this paper does not propose std::execution::with_value.

Note that if `std::execution::task` is extended in the future to support value completions with arities higher than one the addition of `std::execution::with_value` would be indicated then.

Proposal

[execution.syn]

```
template<class E>
  struct with_error {
    using type = remove_cvref_t<E>;
    type error;
  };
template<class E>
  with_error(E) -> with_error<E>;

struct with_stopped {};
```

// [exec.task], class template task

```
template<class T, class Environment>
  class task;
```

[task.state]

```
namespace std::execution {
  template<class T, class Environment>
  template<receiver Rcvr>
  class task<T, Environment>::state {               // exposition only
  public:
    using operation_state_concept = operation_state_tag;

    template<class R>
      state(coroutine_handle<promise_type> h, R&& rr);

    ~state();

    void start() & noexcept;

    stop_token_type get-stop-token();               // exposition only

  private:
```

```
    using own-env-t = see below;                          // exposition only
    using error-t = see below;                            // exposition only
    coroutine_handle<promise_type>      handle;       // exposition only
    remove_cvref_t<Rcvr>                rcvr;         // exposition only
    optional<stop_source_type>          source;       // exposition only
    own-env-t                           own-env;      // exposition only
    Environment                         environment;  // exposition only
    optional<T>                         result;       // exposition only;
                                                      // present only if
                                                      // is_void_v<T> is false
    bool                                stopped{};    // exposition only
    ~~exception_ptr~~std::optional<error-t> error;        // exposition only
  };
}
```

The type *own-env-t* is `Environment::template env_type<decltype(get_env(declval<Rcvr>()))>` if that *qualified-id* is valid and denotes a type, `env<>` otherwise.

The type *error-t* is `variant<Errors...>` where `Errors` is a parameter pack consisting of the `set_error_t` argument types of `error_types`.

[...]

[task.promise]

```
namespace std::execution {
  template<class T, class Environment>
  class task<T, Environment>::promise_type {
  public:
    task get_return_object() noexcept;

    static constexpr suspend_always initial_suspend() noexcept { return {}; }
    auto final_suspend() noexcept;

    void unhandled_exception();
    coroutine_handle<> unhandled_stopped() noexcept;

    void return_void();~~               // present only if is_void_v<T> is true~~
    template<class V = T>
      void return_value(V&& value);~~  // present only if is_void_v<T> is false~~
    template<class E>
      void return_value(with_error<E> error);
```

```
    void return_value(with_stopped stopped);

    template<class E>
      unspecified yield_value(with_error<E> error);

    template<sender Sender>
      auto await_transform(Sender&& sndr);

    unspecified get_env() const noexcept;

    void* operator new(size_t size);
    template<class Alloc, class... Args>
      void* operator new(size_t size, allocator_arg_t, Alloc alloc,
        Args&&...);
    template<class This, class Alloc, class... Args>
      void* operator new(size_t size, const This&, allocator_arg_t,
        Alloc alloc, Args&&...);

    void operator delete(void* pointer, size_t size) noexcept;
  };
}
```

[...]

`auto final_suspend() noexcept;`

Returns: An awaitable object of unspecified type whose member functions arrange for the completion of the asynchronous operation associated with `STATE(*this)`. Let *st* be a reference to `STATE(*this)`. The asynchronous completion first destroys the coroutine frame using `st.handle.destroy()` and then invokes:

- `set_stopped(std::move(st.rcvr))` if `st.stopped` is true, otherwise
- `set_error(std::move(st.rcvr), std::move(`~~`st.error`~~`e))`, where e denotes the active alternative of `*st.error`, if `bool(st.error)` is true, otherwise
- `set_value(std::move(st.rcvr))` if `is_void_v<T>` is true, and otherwise
- `set_value(std::move(st.rcvr), *std::move(st.result))`.

[...]

`void unhandled_exception();`

Effects: If the signature `set_error_t(exception_ptr)` is not an element of `error_types`, calls `terminate()`. Otherwise, ~~stores `current_exception()` into~~evaluates `STATE(*this).error.emplace(current_exception())`.

[...]

```
void return_void();
```

Constraints: `is_void_v<T>` is `true`.

Effects: Does nothing.

```
template<class V>
  void return_value(V&& v);
```

Constraints: `is_void_v<T>` is `false`.

Effects: Equivalent to *`result`*`.emplace(std::forward<V>(v))`.

```
template<class E>
  void return_value(with_error<E> error);
```

Constraints: `is_void_v<T>` is `true` or `with_error<E>` is not convertible to `T`.

Mandates: `std::move(err.error)` is convertible to exactly one of the `set_error_t` argument types of `error_types`.

Effects: Equivalent to *`STATE`*`(*this).`*`error`*`.emplace(std::move(err.error))`.

```
template<class E>
  void return_value(with_stopped stopped);
```

Effects: Equivalent to *`STATE`*`(*this).`*`stopped`*` = true`.

[...]

Acknowledgments

The author would like to thank Dietmar Kühl for bringing the issue addressed hereby to his attention and for providing valuable context regarding the history of and motivation for several `std::execution::task` design decisions.

The author acknowledges that Jonathan Müller badgered him into adding a section discussing a hypothetical `std::execution::with_value`.

References

[1] R. Leahy. return_value & return_void Are Not
Mutually Exclusive P3950R1

[2] D. Kühl et al. Add a Coroutine Task Type P3552R3
[3] L. Baker et al. std::generator: Synchronous Coroutine Generator for Ranges P2168R3